Decentralized Weighted Voting for P2P Data Management[1]

Maya Rodrig, University of Washington
Anthony LaMarca, Intel Research Seattle

IRS-TR-03-004

May 2003

Intel Research

[1] This paper bas been submitted for publication.

Decentralized Weighted Voting for P2P Data Management[1]

Maya Rodrig
University of Washington
Department of Computer Science and Engineering
Seattle, WA 98195-2350
rodrig@cs.washington.edu

Anthony LaMarca
Intel Research Seattle
1100 NE 45th Street
Seattle, WA 98105
lamarca@intel-research.net

ABSTRACT

This paper presents a decentralized variant of David Gifford's classic weighted-voting scheme for managing replicated data. Weighted voting offers a familiar consistency model and supports on-line replica reconfiguration. These properties make it a good fit for applications in the pervasive computing domain. By distributing versioned metadata along with data replicas, and managing access to both data and metadata with the same quorums, our algorithm supports a peer-to-peer environment with dynamic device membership. Our algorithm has been implemented as part of a database called Oasis that was designed for pervasive environments.

1. INTRODUCTION

In this paper we present a decentralized mechanism for providing sequentially-consistent access to data in a partially connected computing environment. This work has been developed in the context of pervasive computing environments. In pervasive computing scenarios it is common for devices to continually arrive and depart and disconnections can be the rule rather than the exception. In these situations, the vast majority of traditional network-oriented services and algorithms will not function properly. This problem is compounded by the heterogeneity and lack of resources available to devices commonly found in these deployments. As a result, the suite of middleware services available in pervasive computing environments is less comprehensive and less evolved than for other distributed computing domains. The lack of software services tailored for pervasive computing forces application developers to address issues such as data management, configuration, and resource allocation for each new application. The burden on application developers is an obstacle to the proliferation of robust and responsive applications in these emerging niches.

As a first step to addressing this challenge, we chose to tackle the data management component of a software services suite for pervasive computing. In order to understand the data management requirements for pervasive applications, we examined fifteen existing applications in the pervasive computing space. Our study showed that while pervasive applications varied widely, there were three things that most of the applications had in common with respect to how they stored, queried and managed data [12]. First, pervasive computing applications typically store structured data and make heavy use of query facilities. Second, the applications have high expectations about data availability, despite the potential for device disconnection or failure. Third, virtually every application assumes that the system infrastructure will be self-managing as users are typically not focused on computing and there is no onsite expertise to guide system management decisions.

To this list, we added a fourth implicit requirement that we believe is critical to the success of pervasive computing. We require that the data management system be easy for programmers to use. Based on this high-level requirement, we identified two specific constraints: First, programmers should not be expected to make explicit data-placements decisions. Second, the system should offer a familiar consistency guarantee, namely sequential consistency. (While there are a variety of weak consistency models that make it easy to ensure availability, as we discuss in Section 2, they are a poor fit for pervasive computing.)

Given this set of application-derived requirements, we developed a P2P meta-database called Oasis [12]. The tension between the requirement for high availability and the need to provide strong consistency guarantees and support disconnected operation became apparent during the design of the system. To address this issue, we adapted a classic consistency mechanism developed by David Gifford called weighted voting [7] to operate within a peer-to-peer architecture. Weighted voting represents a generalization of a basic quorum-based scheme. Weighted voting guarantees sequential consistency [13], supports disconnected operation, and is highly tunable, providing the opportunity for a self-managing system that makes data placement decisions automatically. One drawback of Gifford's weighted voting scheme is the restriction that data be accessed from a single, centralized client. While weighted voting has been used in other consistency algorithms, a decentralized version that preserves the synchronous access and strong consistency of the original has yet to be developed.

Thus, we developed a fully-decentralized variant of Gifford's weighted voting scheme and applied it as the consistency-control mechanism for Oasis. Our algorithm achieves decentralization by storing versioned replicas of the metadata along with the data. To ensure proper metadata and version management, we add an additional constraint to the size of read and write quorums. Finally, to allow replica configurations to be tuned in an online fashion, data and metadata management require the same quorums to be acquired.

The rest of the paper is organized as follows. In Section 2, we discuss related storage management systems and consistency mechanisms. Section 3 describes weighted voting and Section 4 introduces our decentralized variant. In Section 5 we ground our algorithm in a brief discussion of the design and performance of Oasis. Section 6 discusses future work and in Section 7 we conclude.

[1] This paper bas been submitted for publication.

2. RELATED WORK

A wide variety of mechanisms, policies and deployable systems have been developed to provide a set of partially connected clients access to distributed data [2][1][7][10][17]. For our purposes it is useful to position existing mechanisms on two axes: centralized versus decentralized, and pessimistic versus optimistic. Centralized systems are characterized as having distinguished entities that play some critical role; typically these systems will not function if these entities permanently depart. Decentralized solutions capture the current notion of "peer-to-peer" and can tolerate the loss of any reasonable subset of the system and continue to function. Pessimistic systems do not allow data to be updated in multiple disconnected partitions at the same time, ensuring that update conflicts do not occur. Pessimistic systems are conservative in their nature, trading availability for strong consistency. Optimistic systems, on the other hand, allow data to be updated across partitions, and attempt to resolve conflicts when disparate versions are later reconnected. Optimistic systems generally offer weaker consistency guarantees in exchange for higher availability. We briefly highlight systems in the four categories described by these axes and explain why pessimistic decentralized systems are a good fit for pervasive computing.

Gifford's original weighted voting algorithm [7] is an example of a pessimistic, centralized technique. This same space is occupied by the majority of existing distributed file systems work as well [16][18]. These systems typically guarantee sequential consistency and can tolerate a degree of server failure.

Bayou [17] and Coda [10] are both examples of "update anywhere" optimistic, centralized systems. These systems allow data to be updated by disconnected clients and conflicts are resolved in a pair-wise fashion during reconnections. In Bayou updates propagate epidemically and conflicts are resolved by bundling writes with fragments of code called conflict resolvers that represent the application's interest. Coda takes a simpler approach and relies on the user to resolve non-trivial conflicts. Both of these systems employ a special primary copy [1] of the data to commit writes and this distinguished copy earns them the classification of centralized.

The work most closely related to ours is Deno [4] [9], a system that employs an optimistic, decentralized concurrency scheme. Like our system, Deno uses a weighted-voting based concurrency control in a peer-to-peer data management system. Like Bayou, Deno employs an epidemic, pair-wise algorithm to propagate updates between servers. Unlike Gifford's original weighted-voting scheme, Deno's base protocol provides a weak consistency model in which writes are not guaranteed to serialize with reads[2]. One interesting feature in Deno is the ability for a server to give up some of its own votes and create new replicas in an independent fashion.

Finally, our weighted-voting algorithm falls into the pessimistic, decentralized category. The recent focus on peer-to-peer storage system has produced a number of other results in this space. Farsite [3] provides high reliability by replicating data across PCs on a local area network. CFS [5] and OceanStore [11] both aim to build a reliable internet-scale storage solution. These systems have primarily been designed with traditional file-oriented workloads in mind while our work has focused on the pervasive computing domain. This considerably changes the design of the resulting systems.

Pessimistic, decentralized systems are well suited for pervasive computing for two reasons. First, we believe that decentralized techniques have greater potential for ease of management than the centralized schemes do. Pervasive deployments feature high rates of disconnections, device failures and device introduction, and peer-to-peer approaches offer the adaptability and fault tolerance necessary in these types of deployments. Second, based on the application survey we conducted, we do not think that the majority of pervasive computing applications require the update-anywhere semantics offered by optimistic concurrency mechanisms. The flexibility and high availability of optimistic systems typically come with a weaker consistency guarantee. In pervasive computing applications users are often away from screens and input devices making it impossible to perform Coda-esque conflict resolution. While holding theoretical promise, Bayou-style application specific conflict resolvers are famously hard to write. Rollback is another conflict resolution technique that is not applicable in environments like pervasive computing in which physical actuation takes place and cannot be undone. For these reasons, we are interested in applying a conservative approach to pervasive computing and investigating where it works well and where it is inadequate.

3. WEIGHTED VOTING

The traditional way to provide sequential consistency and allow disconnected operation is with a quorum-based scheme in which a majority of the replicas must be present to update the data. We have adapted Gifford's "weighted voting" [7] variant of the basic quorum scheme. In Gifford's scheme, every replica i of a data object is assigned a number of votes V_i. Data replicas are versioned to allow clients to determine which replicas are up to date. The total number of votes assigned to all replicas of the object is N, where $N=\Sigma_i V_i$. An operation must gather a read quorum of R votes to read the object, and a write quorum of W votes to write to the object. Quorums are established by acquiring locks on a set of replicas with a total of at least R or W votes, as needed for reads or writes respectively. Weighted voting ensures sequential consistency by requiring that R+W>N. This constraint guarantees that no read can complete without seeing at least one replica updated by the last write (since R>N-W).

Gifford's weighted voting scheme offers greater flexibility than traditional quorum-based schemes in which each replica is weighted equally. By manipulating the vote allocation, and R and W, Gifford's scheme can be tailored to the expected workload. High performance can be achieved by heavily weighting replicas stored on high performance devices, while reliability can be achieved by allocating more votes to replicas on highly reliable devices. Depending on the ratio of read to write requests to the data object, the relative performance and reliability of reads and writes can be controlled by adjusting R and W. Making R small, for example, increases read performance by allowing clients to be serviced by different replicas in parallel. Making R and W close to N/2 allows up to half the servers to fail, increasing fault-tolerance.

[2] In [4], the authors refer to an extension to Deno's protocols that results in a strong consistency guarantee. We were unable to locate the technical report they referred to and therefore cannot explain how it relates to Oasis.

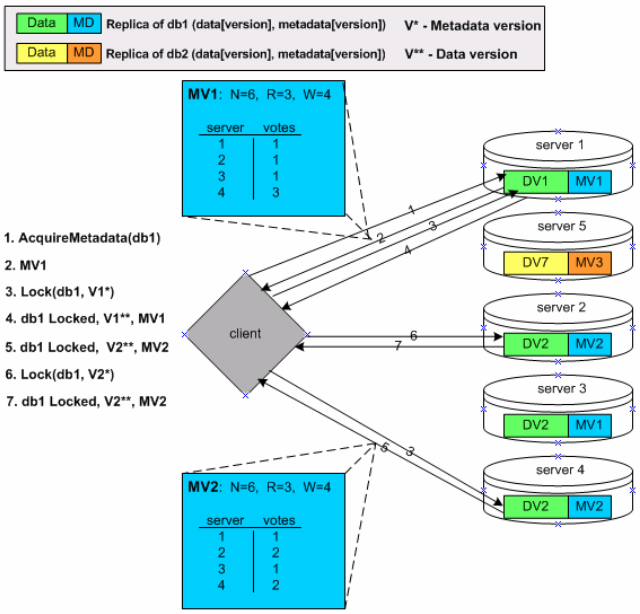


Figure 1 An example of bootstrapping and locking to acquire a write quorum (W=4).

4. DECENTRALIZED WEIGHTED VOTING

The original algorithm presented by Gifford was designed for systems in which replicated data is distributed across multiple servers and accessed by a single client. The client, known as the collector, is used to gather a quorum for every request to the data object. Having a single client simplifies the algorithm by providing a centralized location for the information about the data object, or *metadata* (list of replica locations, versions and vote allocation, R, W, and N), to be maintained. However, maintaining metadata in a centralized fashion is not suitable in a distributed system that experiences frequent device disconnections. As devices come and go, changing the number and placement of the data object's replicas may be required. Gifford's scheme cannot support multiple clients if replica reorganization is allowed and the metadata needs to change.

To manage replicated data in a dynamic distributed system with multiple clients and servers and frequent disconnections we developed a variant of Gifford's weighted voting scheme. We decentralized the functionality of the collector by distributing versioned copies of the metadata along with the data in each replica and providing all clients the ability to access the metadata. By versioning the metadata and imposing the same quorum requirements to update metadata as those to update data, we ensure that quorums are based on the most current metadata, and thus every request accesses the most current data. This allows data and metadata operations to be safely interleaved, enabling the system to perform self-tuning. To guarantee sequential consistency, we enforce the additional constraint that $W \geq R$ (see Section 4.2). With the exception of this additional constraint, our decentralized weighted-voting scheme provides the same flexibility in replica placement and vote assignment as the original.

4.1 The Protocols

In this section we describe the protocols that are central to our decentralized weighted voting scheme. The protocols involve two entities: servers and clients. Servers are storage repositories for replicas of data objects; clients issue read and write requests to the data on behalf of users (applications), thus controlling access to the data and ensuring sequential consistency. Messages are used for communication between clients and servers as well as among servers.

At a high level, our decentralized weighted voting scheme involves three steps. First, a client employs a bootstrapping mechanism to obtain a copy of the metadata for the data object to be read or written. Second, the client uses information found in the acquired metadata to establish a quorum. For write requests, establishing a quorum requires locking a set of replicas with W votes. For read requests, a quorum is obtained by fetching the version of the data object from a set of replicas with R votes and then locking one of the replicas with the most current version of the data. Finally, once the appropriate quorum is established, the client requests the servers storing the locked replicas to execute one of three operations: write data, read data, or update metadata. Once the operation is executed, the replicas involved are unlocked and a reply is sent back to the client. If a write operation fails to complete, a failure handing mechanism is used to ensure the integrity of the data and consistency across replicas.

We begin our protocol description with the bootstrapping and locking mechanisms. We then describe the three operations – write, read, and metadata update – that can be executed once an appropriate quorum is obtained. We end this section with a brief description of failure handing.

4.1.1 Bootstrap

To satisfy a user's read or write request, a client must find metadata for the data object of interest. The client first performs a lookup in its local metadata cache for the desired metadata. If the metadata is found locally, the client can proceed to contact the servers listed in the metadata to acquire a quorum. However, if the metadata is not locally available, the client must determine which servers (potentially storing the desired metadata) are currently available. We assume the existence of a mechanism to locate the locally available servers. The search for servers can be based on a discovery service or one of many distributed search schemes such as DHTs [15]. Once the servers are located, the client contacts them in random order to find the metadata. A more sophisticated search mechanism can replace this random scan when a large number of servers are available (although this was not the case in the pervasive applications we examined). If metadata for the object of interest is not found on any of the locally available servers, the request cannot be processed at that time. If the metadata is found, the bootstrapping process continues.

Based on the acquired metadata, the client selects the servers to contact in order to establish a read or write quorum as needed. The initially acquired metadata may not be up to date. However, since metadata is only updated with a write quorum, at least one of the contacted servers is guaranteed to store an up-to-date version of the metadata. If a newer version of the metadata exists, it is sent back to the client by at least one of the servers. The client may need to contact additional servers to acquire a quorum based on the information in the new metadata. A quorum composed of replicas on servers listed in the current version of the

metadata is guaranteed to include an up-to-date version of the data.

4.1.2 Locking

A client must establish a quorum of votes to access a data object and process the user's request. To acquire the votes, the client contacts the servers storing replicas of the data object (as listed in the metadata) to obtain a lock on their replicas. The client sends a lock request to a server for a specific data object. The server replies with a "yes" or "no", indicating whether the lock was acquired or not, the version of the data object it is storing, and the metadata of the object. If the metadata sent back from the server is newer than the metadata available to the client, the new metadata replaces the older version at the client. New metadata may require the client to send additional lock requests if replica locations, vote distribution, R or W have changed. This is shown in Figure 1 when the client receives a more up-to-date version of the metadata from server 4.

4.1.3 Write Operations

To process a write request a client must first obtain the lock on a set of replicas whose votes sum up to at least W. Once a quorum of W votes is acquired, at least one up-to-date replica is guaranteed to be locked (see Section 4.2). The client selects an up-to-date replica from the quorum and sends a request to the server storing it to update all replicas in the quorum that do not have the most current version of the data. The server contacts the servers holding stale replicas with the list of updates necessary to bring their data replicas up to date. Once the updating server receives a reply from the servers that required updating, it sends a reply back to the client indicating whether all replicas in the quorum were brought up to date. If the reply is positive, the client sends the write request to all the servers holding replicas in the quorum. The servers execute the write request, increment the version of the data, send the result back to the client, and unlock their replicas. Figure 2 shows an example of an update to a collection of servers. If a client cannot acquire a quorum of W votes, the write request cannot proceed.

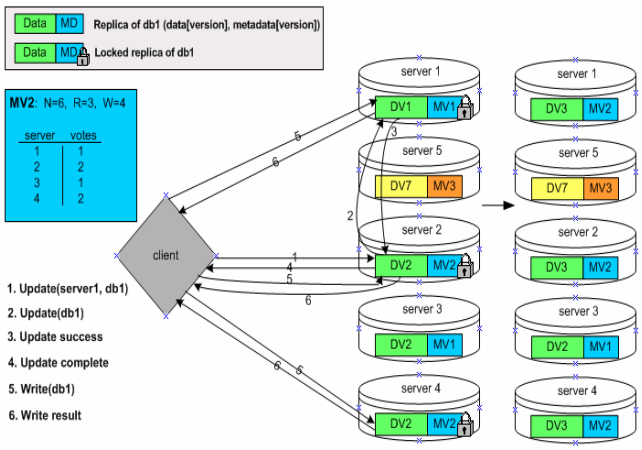


Figure 2 An example of a writer operation following the quorum acquisition. During the operation the servers move from the state shown in the left column to the state shown on the right.

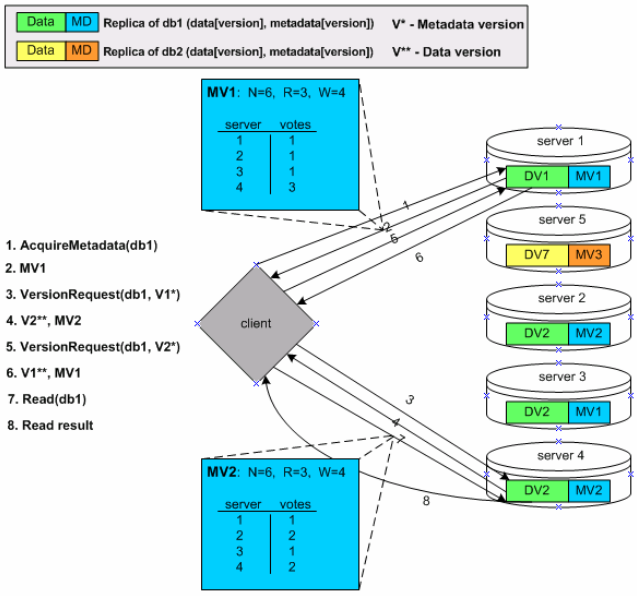


Figure 3 An example of bootstrapping to acquire a read quorum (R=3) and the corresponding read operation.

4.1.4 Read Operations

To process a read request a client must first fetch the versions of the data object from a set of replicas whose votes sum up to at least R. Once the client obtains replies to version requests from a quorum, our algorithm guarantees that at least one of the replies includes the most current version of the data (see Section 4.2). The client sends the read request to one of the servers that responded with the most current version. The server locks its replica, executes the read request, unlocks the replica, and sends the results back to the client. Figure 3 shows an example of a read request being processed.

If a client cannot obtain versions from a quorum of replicas with at least R votes, sequential consistency cannot be guaranteed and thus the read request cannot be processed. These periods of read-unavailability can be eliminated by having a device maintain an up-to-date zero-vote replica. Replicas with zero votes cannot help in the establishment of a quorum, and thus can be added to any device without affecting any other device's access to the data. These local replicas provide a local copy of the data that can always be accessed even if a quorum cannot be acquired. Keeping a local replica up to date requires a change to the read protocol. The server processing the read request piggybacks with the reply a set of changes that can be applied to the client's local copy in order to update it to match the version on the server. The client is required to apply these changes to the 0-vote replica before returning the read result to the user. In this way, even if the client device disconnects, future reads can be serviced from the local copy. While this local copy of the data is potentially stale, sequential consistency is ensured.

4.1.5 Metadata Updates

Updates to the metadata of a data object are either requested by an application or triggered by a self-tuning service that we employ to improve replica placement and vote distribution in the system. Metadata writes are processed much like writes to the data object itself. The same W votes that are required for a quorum to write to

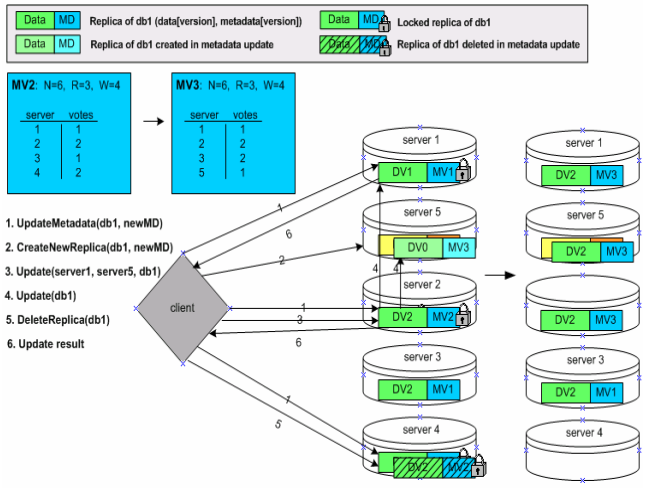


Figure 4 An example of a metadata update following the quorum acquisition. During the operation the servers move from the state shown in the left column to the state shown on the right.

the data, govern updates to the metadata. A write to metadata may change R, W, N, the list of servers storing replicas, or the distribution of votes across replicas. Updates to a data object's metadata must be revealed as changes to its replica configuration in the system. Thus, once a client establishes a write quorum based on replicas in the existing metadata and sends them metadata update requests, some reorganization needs to take place to reflect the migration to the new metadata. Replicas of the data object and the new metadata are sent to servers new on the metadata list, while replicas on servers that are no longer on the metadata list are deleted. Once the changes have been made, the servers participating in the quorum unlock their replicas and send the client a reply (success/failure). Figure 4 shows an example of a metadata update.

4.1.6 Handling Failures

The failure of an update request to complete can render replicas inconsistent. To ensure the consistency of a data object, clients use a two-phase commit protocol when acquiring votes and executing updates on a replica. Two situations may cause an update request to fail: the client issuing the request fails after locking all or part of a quorum, or a server storing a replica fails while the replica is locked as part of a write quorum. In the former case, the servers receive a lock request from the client, but never receive the expected request to update their replicas. After a predetermined period of time, the servers assume that the client has failed while processing the request and invalidate their replicas of the data object. In the latter case, the server invalidates the replica once it recovers from the failure. By making locks persistent, the server knows which replicas were locked at the time of failure and thus require recovery. Invalid replicas cannot participate in client operations until a distributed recovery algorithm [8] has been successfully executed.

4.2 Guarantees

Adapting Gifford's weighted voting algorithm to a P2P environment introduces two challenges beyond ensuring sequentially consistent access to data objects. First, maintaining metadata in a distributed fashion and ensuring the use of up-to-date metadata to establish quorums. Second, propagating version numbers to allow each quorum to determine what the most current version of the data is. In this section we describe the requirements we impose in order to guarantee the use of correct metadata and version propagation.

4.2.1 Correct Metadata

Our decentralized weighted voting scheme relies on the requirement introduced in Gifford's scheme that R+W>N to ensure that every read request accesses the most current version of the data. Since R>N–W, every read quorum is guaranteed to include at least one replica that was part of the last write quorum, and thus have access to up-to-date data. We can provide the same guarantee for accessing metadata since the read and write quorums for accessing metadata must also conform to R+W>N. Guaranteeing access to the most current version of the metadata is necessary in order to ensure that quorums are gathered based on metadata representing the current replica configuration in the system.

During the bootstrapping process, a client may initially acquire metadata that is not up to date and begin sending lock requests to servers listed in that metadata. In rare instances, a client may obtain very stale metadata for which none of the listed servers still hold replicas. In this case, the client must discard the old metadata and re-bootstrap to find a newer version. More commonly, stale metadata leads a client to servers with more and more up-to-date metadata. Having servers piggyback their metadata along with their lock replies guarantees that the client will acquire the current metadata before a quorum is obtained.

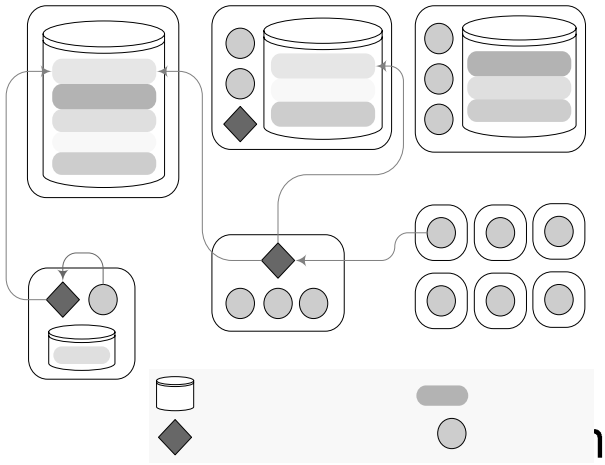
4.2.2 Version Propagation

In Gifford's algorithm, the collector has global knowledge of the version of the data. The replicas that are updated under a write quorum are informed by the collector of their new data version. Even in the case where W<R, in which a write quorum may not include any replica from a write quorum preceding it (since W<N–W), the correct version will be propagated to the replicas in the second write quorum by the collector.

Distributing the metadata and doing away with the centralized client model requires us to enforce one additional constraint on the choice of R and W in our scheme. By adding the requirement W≥R we ensure that W≥R>N–W. As a result, we guarantee that every write quorum includes at least one replica from the write quorum preceding it, and thus the correct version of the data or metadata can propagate from one quorum to the next.

5. Oasis

We have implemented the decentralized weighted voting scheme described in the previous section in the context of a data management system called Oasis. Oasis was designed based on the storage management requirements of a collection of existing pervasive computing applications [12]. In Oasis, the data objects being replicated are relational databases holding some variable number of tables. Both the 'client' and 'server' described in our algorithm have been implemented as services in a pervasive computing framework called Rain [14]. In this framework, services communicate using asynchronous XML messages that are sent over TCP. Applications interact with Oasis by sending a message containing the database name (e.g., 'temperatures') and an SQL query to a weighted voting client. The weighted-voting



client acts on the applications' behalf, executing our algorithm and sending back the results in a reply message. Implementing the client as a separate service rather than a code library requires additional messaging but allows us to support extremely impoverished devices (e.g.: sensor beacons that cannot perform two-way communication). Figure 5 shows an example of an Oasis configuration in an instrumented home.

Our initial implementation of Oasis was written in Java. Server discovery is performed using the discovery service, much like the ones found in Jini and UPnP, provided by Rain [14]. Oasis is a meta-database as it does not actually store and index the data itself, but rather delegates this to an underlying database. For flexibility, the Oasis server has been written to run on top of any JDBC-compliant database that supports transactions. Our initial deployments have used a variety of products: PostgreSQL and MySql have been used on PC-class devices, and PointBase, an embedded, small-footprint database, has been used with IPAQs and other ARM-based devices. In order to support existing applications that use JDBC, we have written a type-4 JDBC driver for Oasis.

In the remainder of this section we discuss several implementation decisions we made that pertain to our weighted voting algorithm. We also present initial experimental data to demonstrate the performance of our weighted-voting based system.

5.1 Data Object Creation

In Oasis, weighted-voting clients handle requests by applications to create new databases. Before a database can be created, it must replica location and vote distribution must be determined. Many real-world applications require a specific replica layout. (e.g.: All the votes go to a primary server, other servers get zero-vote replicas to serve as caches; or 4 servers get one-quarter of the votes each with W=N and R=1 in order to obtain high read performance). Oasis supports explicit configuration by allowing applications to provide the weighted-voting client a specific replica distribution. Oasis also supports a second method in which applications specify expected performance and expected reliability, and Oasis makes the replica configuration decisions itself. Oasis servers describe their own performance and reliability and the weighted-voting clients use a simple additive greedy algorithm to match applications' specifications to the characteristics of available servers in order to decide on the new database's replica configuration. For database creation, a weighted voting client acquires a quorum consisting of all the servers to receive replicas of the new database and send them requests to create the new database.

One interesting problem is how to ensure the uniqueness of database names across an Oasis installation. How can the system determine whether the 'temperatures' database exists but is disconnected, or has never been created? Since our data objects are coarse-grained databases, a simple solution works well. We added a distinguished database that holds the names of all the databases in the system. If a new database is to be created, it must first be added uniquely to this distinguished database. While this implies that databases cannot be created in multiple locations at the same time, it would not present a problem for the applications we examined in our study; In general, databases were created at installation time only. In order to support the creation of temporary 'scratch' storage, Oasis can create databases with random unique names without requiring that the distinguished database be available for writing. The replica and vote layout of this distinguished database can be adjusted to suit the requirements of the applications.

5.2 Locking for Read Operations

The protocol for the read operation described in Section 4.1.4 requires a client to lock a single replica with the current version of the data once it has obtained replies to version requests from a set of replicas whose votes sum to R. This lock request can be eliminated completely if the underlying data object (database, file, record, etc.) supports transactional updates. In this case, updates are atomic, ensuring that any read request sees either the value before or after an update, and not some invalid state in between.

5.3 Avoiding Deadlocks

Clients are required to lock a set of replicas to update a data object. If multiple clients attempt to update a data object at the same time, their lock requests may interleave in such a way that each client acquires the lock on a number of replicas, and gets queued for the lock of at least one replica. Thus, no client is able to obtain a full quorum (W votes) in order to process the write request, resulting in deadlock. The simplest way to eliminate deadlock is to establish a total order on the servers and require that they be locked in this order. While this ensures correctness, it eliminates the parallelism that was originally available in the lock-acquisition process. In order to avoid deadlock while exploiting parallelism when possible, Oasis clients track the contention for a database. If contention is low, a client may try to acquire locks in parallel. However, if one of the lock requests gets queued by any of the servers, the client releases the locks it had acquired and switches to a sequential, ordered lock acquisition.

5.4 Partial Updates

In a write operation, all replicas in the write quorum must be brought up to date before the new update can be applied. This update can be applied by either shipping the updated object or by shipping a set of delta queries that will bring the server's replica up to date. Oasis servers maintain a sliding window of recent queries, and if a partial update can be achieved using this set of deltas the partial update is favored over the full update.

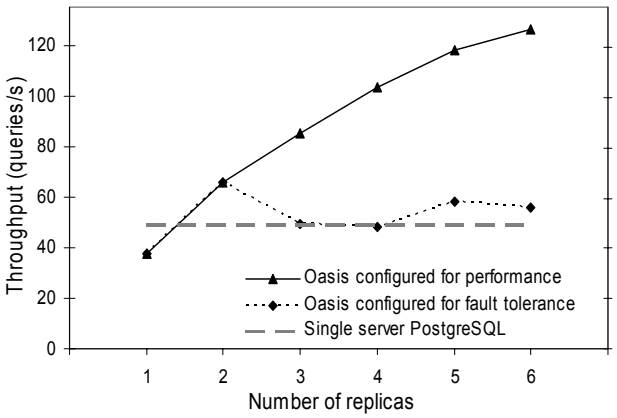


Figure 6 This graph compares the throughput of two Oasis configurations and a single PostgreSQL server.

5.5 Permanent Failures

One of the key features of our weighted voting scheme is support for disconnected operation. A replica on a mobile device can be assigned the majority of the votes to allow reads and writes to the data while the device is disconnected. While disconnection may be the norm in some applications, it is important to distinguish between disconnections and permanent failures. If a device fails, the replicas it stores become inaccessible and can no longer participate in quorums for read or write operations. To ensure that inaccessible replicas do not render data objects unavailable for prolonged periods of time, a lease mechanism to invalidate replicas is necessary. Oasis assigns each replica a lease period and provides a mechanism to reallocate votes among replicas. The lease period needs to be sufficiently long in order to avoid invalidating a replica too quickly. We have not yet implemented the mechanism to trigger the shift of votes away from a timed-out replica.

5.6 Performance

Oasis has been used by other researchers in our laboratory to implement a variety of pervasive computing applications. The application with the most stringent performance requirements is called Guide [6]. Guide is an investigation into the use of radio-frequency ID tags (rfid tags) and robots to track the location of physical objects in the environment. To measure the performance of Oasis, we ran an experiment using the database schemas from the Guide application. The Guide database is comprised of 3 tables: a *reading* table tracking when and where an rfid-tag was seen, an *object* table that relates rfid-tags to object names (like 'stapler'), and finally a *place* table that records the geometric bounds of rooms and locations. Our experimental data set was seeded with 1 million records in the reading table, 1000 records in the object table and 25 records in the place table. (This is the number of objects we expect to tag in our laboratory and the number of readings we expect to collect and in a month.)

In our test a set of weighted-voting clients alternate between performing reads and writes to the database. The read queries are all of the form "Where was object X last seen" and require a join across the reading and place tables. The updates are all insertions of a new rfid reading into the reading table. The clients perform reads and writes in a 50:1 ratio, as we would expect in a real Guide deployment. For this workload, we measured three database configurations. As a control, we ran a single, non-replicated database server of the same type that Oasis uses as the underlying database (PostgreSQL). We also ran two Oasis configurations, one set up for maximum read throughput (R=1, W=N), the other for maximum fault-tolerance (R=N/2, W=N/2+1). The high-performance configuration cannot tolerate any server failures, while in the fault-tolerant configuration, up to $\lceil N/2 \rceil - 1$ replicas can fail or disconnect without affecting clients' access to the data. In our experiments, the number of clients is fixed at 10 and the number of replicas is varied from 1 to 6. Each client and server in the test ran on its own Pentium 4 PC running either Windows 2000 or Linux connected via 100MB/s Ethernet. The Oasis servers and clients ran on Sun's 1.3.1 JVM and the underlying data was stored in PostgreSQL 7.3.

Figure 6 shows the total throughput achieved by the set of weighted-voting clients. As expected, the weighted voting algorithm adds overhead, resulting in lower performance than PostgreSQL when Oasis is configured with a single replica. However, as replicas are added to Oasis, read parallelism can be exploited and the read-dominated workload can be spread across multiple servers. This effect is most pronounced in the case when W=N and R=1 where any replica can independently service a read query. Even in the fault-tolerant configuration, read parallelism exists and for all multiple-replica configurations Oasis outperforms the single PostgreSQL server. The lack of smoothness in the curve for the fault-tolerant configuration is due to two factors. First, when R=1, there is no need to first collect version numbers and then send the lock/query request. Since any replica can be read from, we can satisfy the read with a single message roundtrip. This optimization can no longer be applied when the third replica is added in the fault tolerant case (N=3, R=2, W=2), resulting in a noticeable drop in throughput. Second, from that point on, one extra up-to-date replica is available for reading for every 2 replicas added, creating the staircase effect.

This graph also shows that Oasis exhibits more than reasonable performance for the average pervasive computing application. The following table shows a breakdown of the time spent performing the Guide workload against an unloaded, 2-way replicated Oasis database (N=2, R=1, W=2).

	Read	**Write**
Locking	0.6 ms	0.6 ms
Messaging	7.0 ms	17.6 ms
Query execution	22.7 ms	10.3 ms
Total	*30.3 ms*	*30.5 ms*

As expected, the complexity of the read queries results in higher query execution time than the write queries. The overhead introduced by Oasis includes locking and messaging costs. The cost of managing the two-phase locking is small; however, the messaging overhead is quite substantial. (The writes have twice the messaging overhead since they require two message roundtrips compared to one for the reads.) The Rain toolkit uses XML to communicate between the Oasis clients and servers. The actual network and socket overhead should be less than a millisecond and we believe that the main factor in the messaging overhead is the cost of parsing the XML messages. This suggests

that using an optimized message transport could offer significant performance gains.

6. FUTURE WORK

The largest practical drawback to using Oasis is the requirement that databases be fully replicated. Currently, an application using Oasis that wants to replicate a part of a database on an impoverished device must create an alternate database and manually keep it consistent. A number of partial replication alternatives exist that allow data to be replicated at the table, record, or SQL-view granularity. We plan to investigate what changes our APIs and consistency mechanism would require in order to support various partial replication schemes.

Recent history suggests that developing application specific conflict resolvers are beyond the ability of average software developers. Despite the potential benefits and the availability of systems that use these resolvers, no real applications have been developed to use them. We would like to explore the potential of *domain specific conflict resolvers* that would represent the usage patterns of classes of applications. (Examples of application classes include: loggers, list managers, configurers, etc.) In this way, we may be able to leverage the power of systems like Deno [4] and Bayou [17] that support smart conflict resolution without placing undue burden on programmers.

Finally, one of the reasons we chose weighted voting was for its flexibility in configuration. In order to see if this flexibility can be exploited to create a self-tuning, self-managing storage system, we need to explore how existing machine-leaning techniques can improve replica management in Oasis. By tracing application data usage and device migration over time, we hope to build a system that can make good automated decisions on where data should be placed, as well as how it should be indexed.

7. CONCLUSION

Balancing programmability and availability presents a significant challenge to data management solutions in partially-connected computing environments. In our work we have taken a conservative approach, maintaining strong consistency guarantees and trying to improve availability and performance within those bounds. To achieve these goals and minimize management overhead, we have developed a decentralized weighting-voting algorithm that guarantees sequential consistency. Our algorithm distributes versioned metadata along with the data and allows online reconfiguration by using the same quorums to manage both the data and the metadata. Our consistency algorithm has been implemented in the context of Oasis, a meta-database designed for pervasive computing environments. Our initial experience with Oasis suggests that its feature set fits both the functional and performance requirements of pervasive computing applications.

8. REFERENCES

[1] Barbara, D., Garcia-Molina, H., Replicated Data Management in Mobile Environments: Anything New Under the Sun? In IFIP Working Conference on Applications in Parallel and Distributed Computing, April 1994.

[2] Amir, Y., and Wool, A. Evaluating Quorum Systems over the Internet. In The Fault-Tolerant Computing Symposium (FTCS) (June 1996), pp. 26--35.

[3] Bolosky, W., Douceur J., Ely, D. and Theimer M., Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs", In Proceedings of ACM Sigmetrics, 2000.

[4] Cetintemel, U., Keleher, P. J., Franklin, M. J., Support for Speculative Update Propagation and Mobility in Deno. In Proc. of 21st IEEE International Conference on Distributed Computing Systems, 2001.

[5] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In Proceedings of the 18th ACM Symposium on Operating Systems Principles, 2001.

[6] Guide, http://seattleweb.intel-research.net/projects/guide/, visited March 2003.

[7] Gifford, D. K., Weighted Voting for Replicated Data, Proceedings of the Seventh Symposium on Operating Systems Principles, 1979, pp. 150-162.

[8] Goodman, N., Skeen, D., Chan, A., Dayal, U., Fox, S, and Ries, D., A recovery algorithm for a distributed database system, in Proceedings 2nd ACM Symposium on Principles of Database Systems, March, 1983.

[9] Keleher, P., Decentralized Replicated-Object Protocols. In Proc. 18th ACM Symp. on Principles of Distributed Computing, (1999), 143-151.

[10] Kistler, J., Satyanarayanan, M. Disconnected Operation in the Coda File System. ACM Transactions on Computer Systems, Feb. 1992.

[11] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B., OceanStore: An Architecture for Global-Scale Persistent Storage, ASPLOS, 2000.

[12] LaMarca, A., Rodrig, M. Oasis: An Architecture for Simplified Data Management and Disconnected Operation, Intel Research Seattle Technical Report IRS-TR-03-003, May. 23, 2003.

[13] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocessor programs. IEEE Trans. on Computers, 28(9):690-691, Sept. 1979.

[14] Rain, http://seattleweb.intel-research.net/projects/rain/, visited May 2003.

[15] Stoica, I, Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications, ACM SIGCOMM 2001, San Deigo, CA, August 2001, pp. 149-160

[16] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS File System. In Proc. of the 1996 Winter USENIX, 1996, 1-14.

[17] Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M. and Hauser, C. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System", Proc. 15th ACM Symp on Operating Systems Principles, (1995), 172-183.

[18] Thekkath, C., T. Mann, and E. Lee. Frangipani: A Scalable Distributed File System. In 16th ACM Symposium on Operating Systems Principles, 1997, 224-237.

[19] Yang, B., and Garcia-Molina, H. Designing a super-peer network. Technical Report, Stanford University, February 2002.